加微信：642945106 发送“ 赠送 ”领取赠送精品课程
发数字“ 2 ”获取众筹列表

11 | Java编译器（三）：属性分析和数据流分析

2020-06-26 宫文学

编译原理实战课

进入课程 >

讲述：宫文学
时长 21:24　大小 19.62M

你好，我是宫文学。

在上一讲，我们主要讨论了语义分析中的 ENTER 和 PROCESS 阶段。今天我们继续往下探索，看看 ATTR 和 FLOW 两个阶段。

ATTR 的字面意思是做属性计算。在第 4 讲中，我已经讲过了属性计算的概念，你应该还记得什么是 S 属性，什么是 I 属性。那么，Java 编译器会计算哪些属性，又会如何计算呢？

FLOW 的字面意思是做数据流分析。通过第 7 讲，你已经初步了解了数据流分析的算法。但那个时候是把数据流分析用于编译期后端的优化算法，包括删除公共子表达式、变量

传播、死代码删除等。而这里说的数据流分析，属于编译器前端的工作。那么，前端的数据流分析会做什么工作呢？

这些问题的答案，我今天都会为你一一揭晓。好了，我们进入正题，首先来看看 ATTR 阶段的工作：属性分析。

ATTR：属性分析

现在，你可以打开 **com.sun.tools.javac.comp.Attr 类**的代码。在这个类的头注释里，你会发现原来 ATTR 做了四件事，分别在 4 个辅助类里实现：

1. Check：做类型检查。
2. Resolve：做名称的消解，也就是对于程序中出现的变量和方法，关联到其定义。
3. ConstFold：常量折叠，比如对于“2+3”这种在编译期就可以计算出结果的表达式，就直接计算出来。
4. Infer：用于泛型中的类型参数推导。

我们首先来看 Check，也就是类型检查。

类型检查

类型检查是语义分析阶段的一项重要工作。静态类型系统的语言，比如 Java、C、Kotlin、Swift，都可以通过类型检查，避免很多编译错误。

那么，一个基础的问题是：**Java 都有哪些类型?**

你是不是会觉得这个问题挺幼稚？Java 的类型，不就是原始数据类型，再加上类、接口这些吗？

说得对，但是并不全面。你已经看到，Java 编译器中每个 AST 节点都有一个 type 属性。那么，一个模块或者一个包的类型是什么？一个方法的类型又是什么呢？

在 java.compile 模块中，定义了 Java 的语言模型，其中有一个包，是对 Java 的类型体系做了设计，你可以看一下：

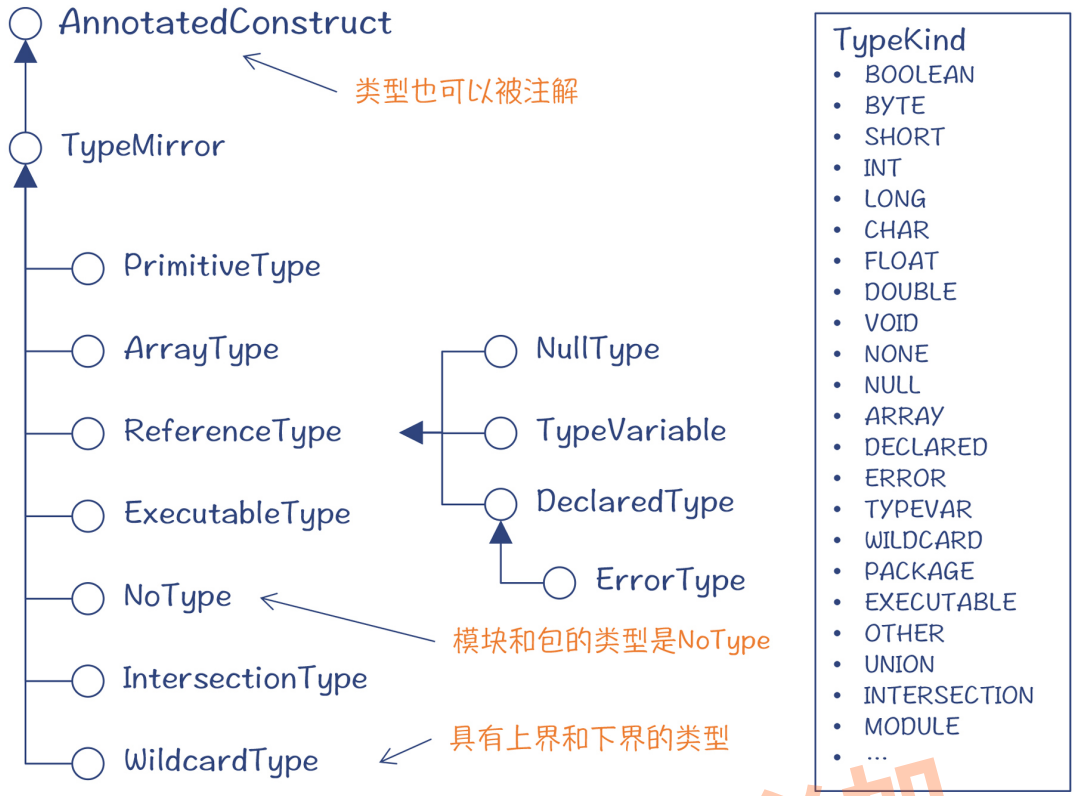


图 1：Java 的类型体系

防止断更 请务必加
首发微信：1716143665

这样你就能理解了：原来模块和包的类型是 NoType，而方法的类型是可执行类型（ExecutableType）。你可以看一下源代码，会发现要刻画一个可执行类型是比较复杂的，竟然需要 5 个要素：

returnType：返回值类型；

parameterTypes：参数类型的列表；

receiverType：接收者类型，也就是这个方法是定义在哪个类型（类、接口、枚举）上的；

thrownTypes：所抛出异常的类型列表；

typeVariables：类型参数的列表。

如果你学过 C 语言，你应该记得描述一个函数的类型只需要这个列表中的前两项，也就是返回值类型和参数类型就可以了。通过这样的对比，想必你会对 Java 的可执行类型理解得更清楚。

然而，通过一个接口体系来刻画类型还是不够细致，Java 又提供了一个 TypeKind 的枚举类型，把某些类型做进一步的细化，比如原始数据类型进一步细分为 BOOLEAN、BYTE、SHORT 等。这种设计方式可以减少接口的数量，使类型体系更简洁。你也可以在编程中借鉴这种设计方式，避免产生过多的、没有什么实际意义的子类型。

同样，在 jdk.compiler 模块中，有一些具体的类实现了上述类型体系的接口：

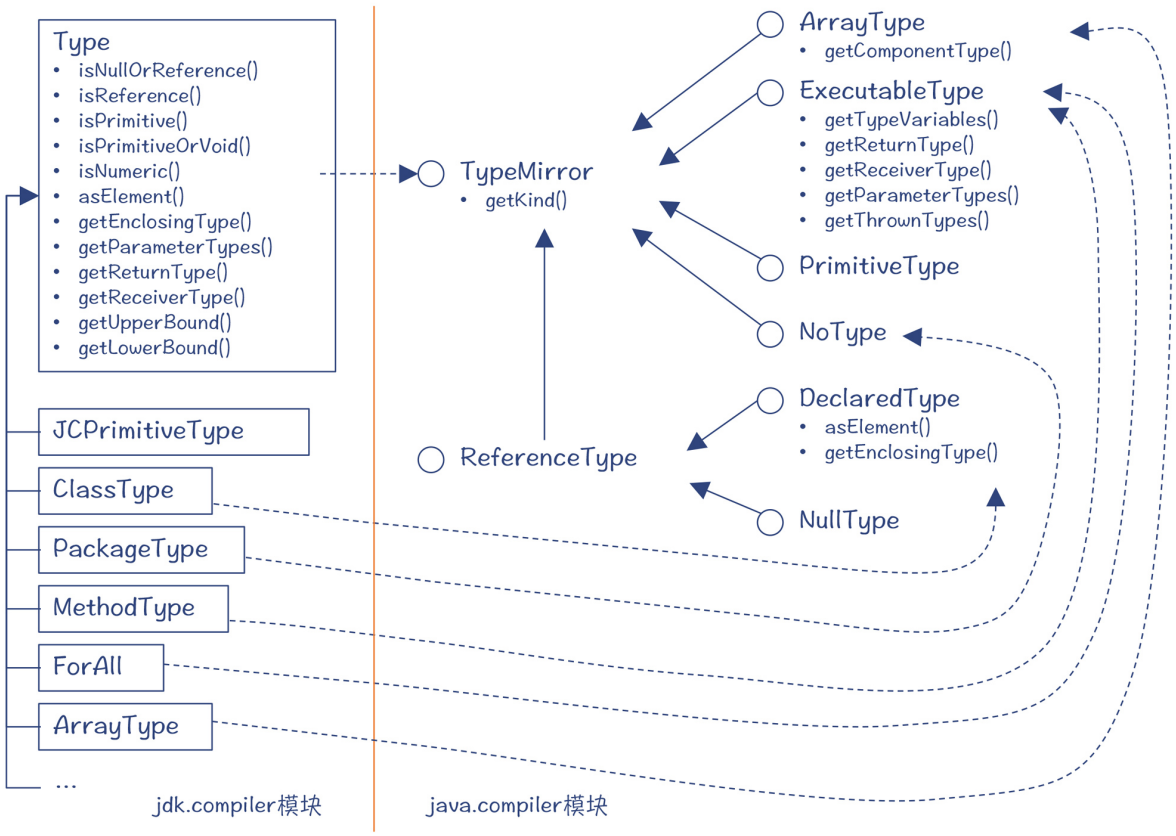


图 2：类型体系的实现

好了，现在你已经了解了 Java 的类型体系。**那么，编译器是如何实现类型检查的呢?**

我用一个 Java 程序的例子，来给你做类型检查的说明。在下面这段代码中，变量 a 的声明语句是错误的，因为等号右边是一个字符串字面量“Hello”，类型是 java.lang.String，跟变量声明语句的类型“int”不相符。在做类型检查的时候，编译器应该检查出这个错误来。

而后面那句“`float b = 10`”，虽然变量 b 是 float 型的，而等号右边是一个整型的字面量，但 Java 能够自动把整型字面量转化为浮点型，所以这个语句是合法的。

```
public class TypeCheck{
    int a = "Hello"; //等号两边的类型不兼容，编译报错
    float b = 10;    //整型字面量可以赋值给浮点型变量
}
```

对于“int a = "hello"”这个语句，它的类型检查过程分了四步，如下图所示：

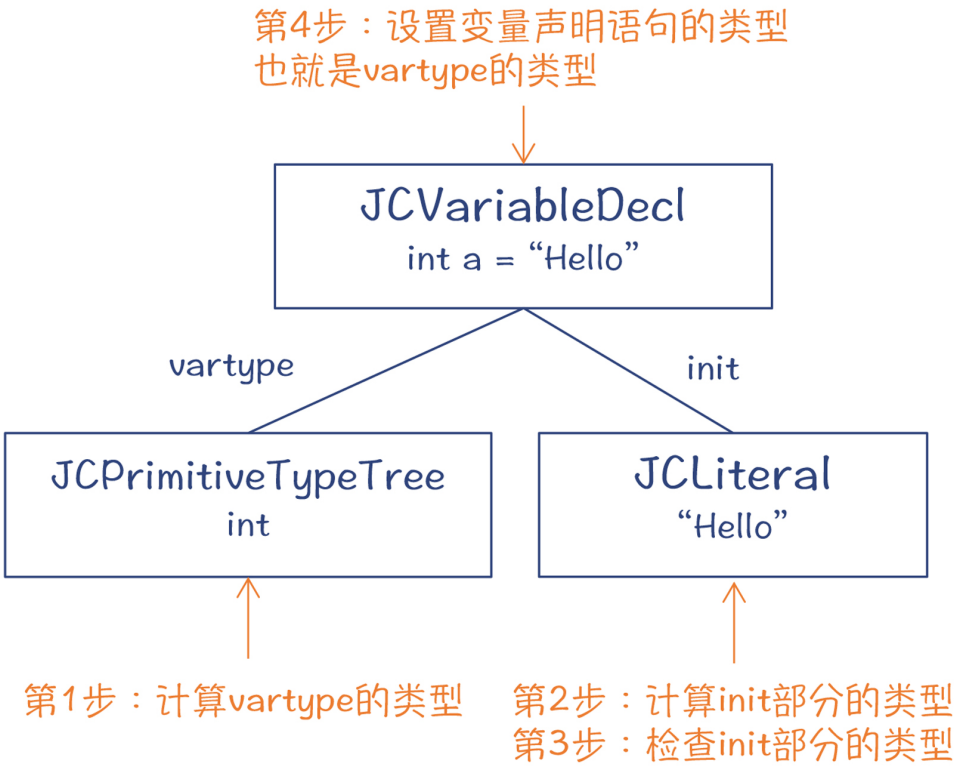


图 3：类型检查的过程

第 1 步，计算 vartype 子节点的类型。这一步是在把 a 加入符号表的时候（MemberEnter）就顺便一起做了（调用的是“Attr.attribType() 方法”）。计算结果是 int 型。

第 2 步，在 ATTR 阶段正式启动以后，深度优先地遍历整棵 AST，自底向上计算每个节点的类型。自底向上是 S 属性的计算方式。你可以看一下 Attr 类中的各种 attribXXX() 方法，大多数都是要返回一个类型值，也就是处理完当前子树后的类型。这个时候，能够知道 init 部分的类型是字符串型（java.lang.String）。

第 3 步，检查 init 部分的类型是否正确。这个时候，比对的就是 vartype 和 init 这两棵子树的类型。具体实现是在 Check 类的 **checkType()** 方法，这个方法要用到下面这两个参数。

> **final Type found**：“发现”的类型，也就是“Hello”字面量的类型，这里的值是 java.lang.String。这个是自底向上计算出来的，属于 S 属性。
>
> **final Type req**：“需要”的类型，这里的值是 int。也就是说，a 这个变量需要初始化部分的类型是 int 型的。这个变量是自顶向下传递下来的，属于 I 属性。

所以你能看出，所谓的类型检查，就是所需类型（I 属性）和实际类型（S 属性）的比对。

这个时候，你就会发现类型不匹配，从而记录下错误信息。

下面是在做类型检查时整个的调用栈：

复制代码

```
JavaCompiler.compile()
->JavaCompiler.attribute()
->Attr.attib()
->Attr.attribClass()       //计算TypeCheck的属性
->Attr.attribClassBody()
->Attr.attribStat()        //int a = "Hello";
->Attr.attribTree()        //遍历声明成员变量a的AST
->Attr.visitVarDef()       //访问变量声明节点
->Attr.attribExpr(TCTree,Env,Type)//计算"Hello"的属性,并传入vartype的类型
->Attr.attribTree()        //遍历"Hello"AST，所需类型信息在ResultInfo中
->Attr.visitLiteral()      //访问字面量节点，所需类型信息在resultInfo中
->Attr.check()             //把节点的类型跟原型类型(需要的类型)做比对
->Check.checkType()        //检查跟预期的类型是否一致
```

第 4 步：继续自底向上计算类型属性。这个时候会把变量声明语句 JCVariableDecl 的类型设置为 vartype 的类型。

上面是对变量 a 的声明语句的检查过程。对于“`float b = 10`”的检查过程也类似，但整型是允许赋值给浮点型的，所以编译器不会报错。

说完了类型检查，我们继续看一下 Resolve，也就是引用的消解。

引用消解

在第 5 讲中，我就介绍过了引用消解的概念。给你举个例子，当我们在程序中用到一个变量的时候，必须知道它确切的定义在哪里。比如下面代码中，第 4 行和第 6 行都用到了一个变量 a，但它们指的不是同一个变量。**第 4 行的 a 是类的成员变量，第 6 行的 a 是 foo() 函数中的本地变量。**

复制代码

```
1  public class RefResolve extends RefResolveParent {
2      int a = 2;
3      void foo(int d){
4          int b = a + f;    //这里的a是RefResolve的成员变量
5          int a = 3;        //本地变量a,覆盖了类的成员变量a
6          int c = a + 10;   //这里的a是前一句中声明的本地变量
7      }
8  }
9
10 class RefResolveParent{
11     int f = 4;           //父类中的成员变量
12 }
```

在编译器中，这两行中的 a 变量，都对应一个标识符（JCIdent）节点，也都会关联一个 Symbol 对象。但这两个 Symbol 对象不是同一个。第 4 行的 a 指的是类的成员变量，而第 6 行的 a 指的是本地变量。

所以，具体到 Java 编译器，引用消解实际上就是把标识符的 AST 节点关联到正确的 Symbol 的过程。

引用消解不仅仅针对变量，还针对类型、包名称等各种用到标识符的地方。如果你写了“System.out.println()”这样一个语句，就要引用正确的包符号。

你可以打开 com.sun.tools.javac.comp.Resolve 类的 **findIdentInternal 方法**，能看到对几种不同的符号做引用消解的入口。

复制代码

```
1  ...
2
3  if (kind.contains(KindSelector.VAL)) {    //变量消解
4      sym = findVar(env, name);
5      ...
```

```
}

if (kind.contains(KindSelector.TYP)) {    //类型消解
    sym = findType(env, name);
    ...
}

if (kind.contains(KindSelector.PCK))      //包名称消解
    return lookupPackage(env, name);
...
```

引用消解的实现思路也很清晰。在上一讲，你知道编译器在 Enter 阶段已经建立了作用域的嵌套结构。那么在这里，**编译器只需要沿着这个嵌套结构逐级查找就行了。**

比如，对于“int b = a + f”这个变量声明语句，在查找变量 a 时，沿着 Scope 的嵌套关系往上查找两级就行。但对于变量 f，还需要沿着类的继承关系，在符号表里找到父类（或接口），从中查找有没有名称为 f 的成员变量。

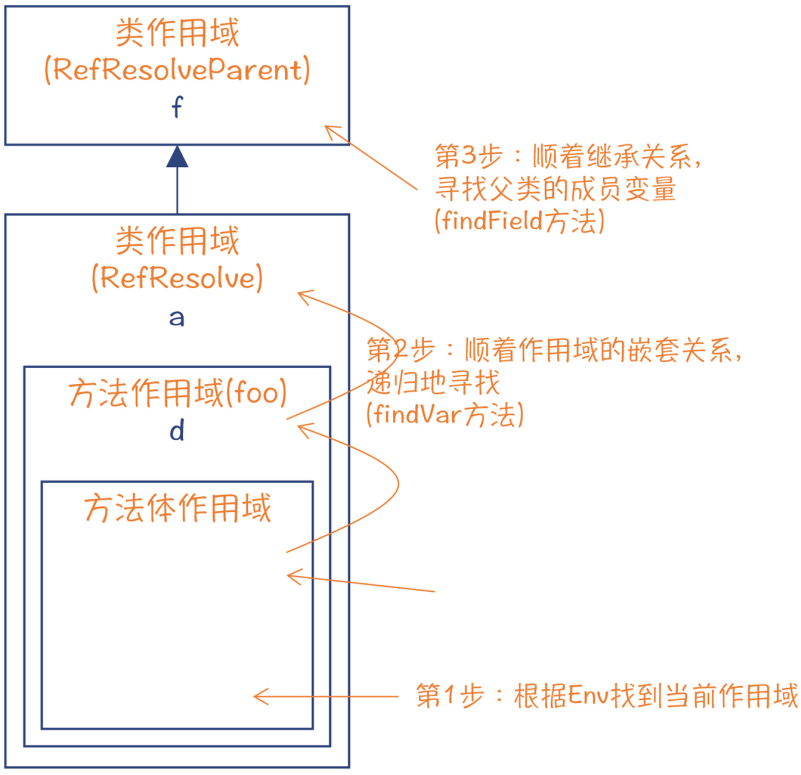


图 4：引用消解的实现

不过，这里还有一个细节需要深究一下。还记得我在前一讲留了一个问题吗？这个问题是：**对于方法体中的本地变量，不是在 ENTER 阶段创建符号，而是在 ATTR 阶段。**具体来

说，就是在 ATTR 的 Resolve 环节。这是为什么呢？为什么不在 ENTER 环节把所有的符号都识别出来，并且加到作用域中就行了？

我来解答一下这个问题。我们把 RefResolve 类中的“int a = 2;”这行注释掉，会发生什么事情呢？foo() 函数的第一行“int b = a + f”应该报错，因为找不到 a 的定义。

复制代码

```
public class RefResolve extends RefResolveParent{
    //int a = 2;           //把这行注释掉
    void foo(int d){
        int b = a + f;    //这里找不到a，应该报错
        int a = 3;         //本地变量a,覆盖了类的成员变量a
        int c = a + 10;    //这里的a是前一句中声明的本地变量
    }
}
```

但是，如果编译器在 ENTER 阶段就把所有的符号建立起来了，**那么会发生什么情况呢**？foo() 的方法体所对应的 Scope 就会有一个符号 a。按照前面描述的逐级查找算法，它就会认为“int b = a + f”里的这个 a，就是本地变量 a。这当然是错误的。

所以，为了保证消解算法不出错，必须保证在做完“int b = a + f”这句的引用消解之后，才会启动下一句“int a = 3”的 ENTER 过程，把符号 a 添加的 foo() 方法体的作用域中。引用消解都处理完毕以后，符号表才会填充完整，如下图所示：

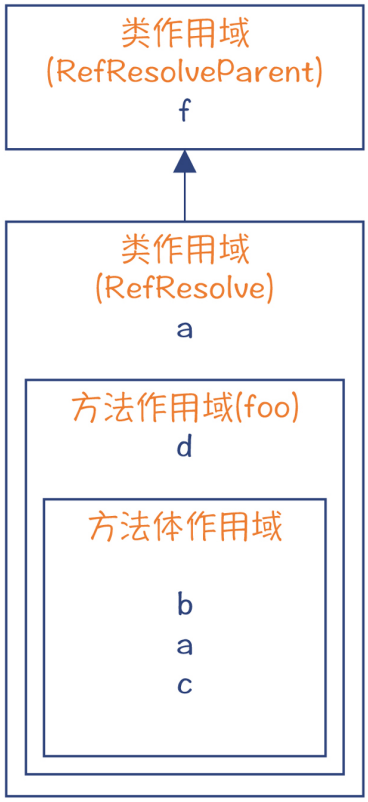


图 5：引用消解后，符号表中添加了本地变量

常数折叠

在 ATTR 阶段，还会做一项优化工作：Constant Fold，即常数折叠。

我们知道，优化工作通常是在编译器的后端去做的。但因为 javac 编译器只是个前端编译器，生成字节码就完成任务了。不过即使如此，也要保证字节码是比较优化的，减少解释执行的消耗。

因为常数折叠借助属性计算就可以实现，所以在 ATTR 阶段顺便就把这个优化做了。

Java 在什么情况下会做常数折叠呢？我们来看看下面这个例子。变量 a 和 b 分别是一个整型和字符串型的常数。这样的话，“c=b+a*3”中 c 的值，是可以在编译期就计算出来的。这要做两次常数折叠的计算，最后生成一个“Hello 6”的字符串常数。

复制代码

```
public class ConstFold {
    public String foo(){
        final int a = 2;           //int类型的常数
        final String b = "Hello "; //String类型的常数
```

```
5           String c = b + a * 3;      //发生两次折叠
6           return c;
7       }
8   }
```

触发上述常数折叠的代码，在 com.sun.tools.javac.comp.Attr 类的 **visitBinary() 方法**中，具体实现是在 com.sun.tools.javac.comp.ConstFold 类。它的计算逻辑是：针对每个 AST 节点的 type，可以通过 Type.constValue() 方法，看看它是否有常数值。如果二元表达式的两个子节点都有常数值，那么就可以做常数折叠，计算出的结果保存在父节点的 type 属性中。你可以看看下图。

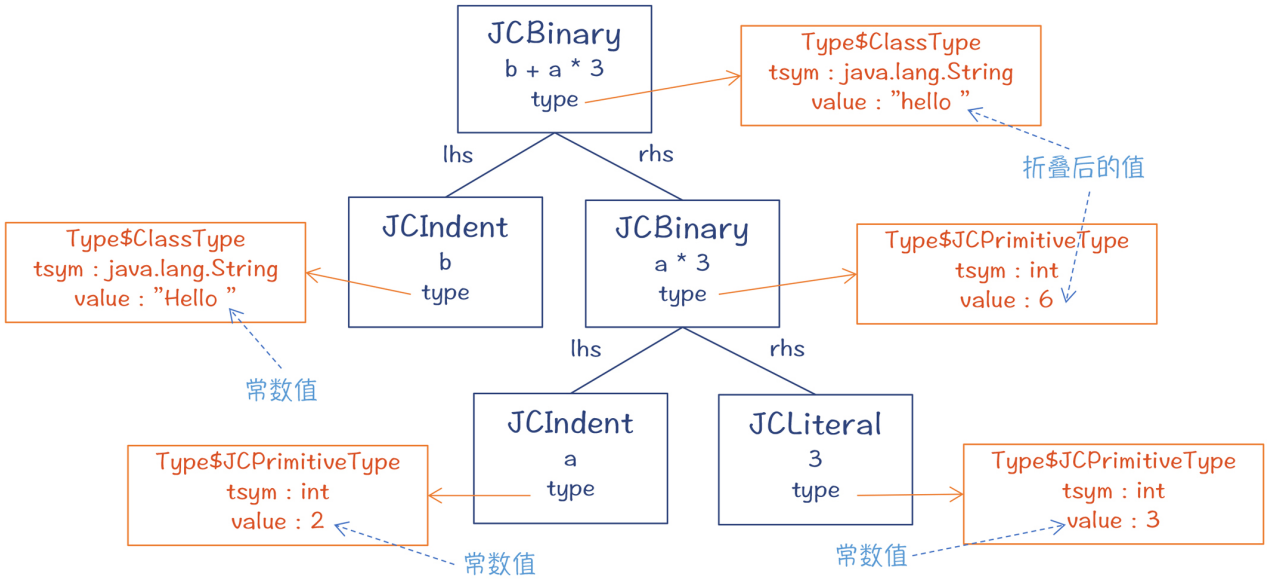


图 6：AST 节点对应的常数值属性

扩展：你看了这个图，可能会有一个疑问：常数值为什么不是保存在 AST 节点中，而是保存在类型对象中？类型带上一个值是什么意思？常数值为 2 的整型和常数值为 3 的整型是不是一个类型？

这是因为 Type 里保存的信息本来就比较杂。我们前面分析过，一个可执行类型（比如方法）里包含返回值、参数类型等各种信息。一个类型的元数据信息（通常指标注），也是存在 Type 里面的。所以，一个方法的类型信息，跟另一个方法的类型信息，是迥然不同的。在这里，不要把 Type 叫做“类型”，而是叫“类型信息”比较好。每个类型信息对象只针对某个 AST 节点，包含了该节点与类型有关的各种信息。因此，在这里面再多放一个常数值，也就无所谓了。

你能看出，常数折叠实质上是针对 AST 节点的常数值属性来做属性计算的。

推导类型参数

ATTR 阶段做的最后一项工作，也是跟类型相关，那就是对泛型中的类型参数做推导。

这是什么意思呢？在 Java 语言中，如果你前面声明了一个参数化类型的变量，那么在后面的初始化部分，你不带这个参数化类型也是可以的，编译器会自动推断出来。

比如下面这句：

```
List<String> lines = new ArrayList<String>();
```

复制代码

你可以去掉初始化部分中的类型参数，只保留一对尖括号就行了：

```
List<String> lines = new ArrayList<>();
```

复制代码

甚至更复杂的参数化类型，我们也可以这样简化：

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
//简化为:
Map<String, List<String>> myMap = new HashMap<>();
```

复制代码

你可以在 Infer.instantiateMethod() 方法中打个断点，观察一下泛型的推断。关于泛型这个主题，我会在“揭秘泛型编程的实现机制”这一讲，去展开讲一些关于类型计算的算法，这里就不详细展开了。

好了，到这里，你已经知道了属性分析所做的四项工作，它们分别针对了四个属性：

类型检查针对的是类型属性；

引用消解是针对标识符节点的符号（sym）属性，也就是要找到正确的符号定义；

常数折叠针对的是常数值属性；

类型参数的推导，针对的是类型参数属性。

所以，现在你就可以解答出学教科书时通常会遇到的一个疑问：属性计算到底是计算了哪些属性。我们用实战知识印证了理论 。

接下来我们看看编译器下一个阶段的工作：数据流分析。

FLOW：数据流分析

Java 编译器在 FLOW 阶段做了四种数据流分析：活跃性分析、异常分析、赋值分析和本地变量捕获分析。我以其中的活跃性分析方法为例，来给你做讲解，这样其他的几个分析方法，你就可以举一反三了。

首先，我们来看看活跃性分析方法对 return 语句的检测。

举个最简单的例子。下面这段代码里，foo 函数的返回值是 int，而函数体中，只有在 if 条件中存在一个 return 语句。这样，代码在 IDE 中就会报编译错误，提示缺少 return 语句。

复制代码

```
public class NoReturn{
    public int foo(int a){  //在a<=0的情况下，不会执行return语句
        if (a> 0){
            return a;
        }
        /*
        else{
            return -a;
        }
        */
    }
}
```

想要检查是否缺少 return 语句，我们就要进行活跃性分析。活跃性分析的具体实现是在 Flow 的一个内部类 LiveAnalyzer 中。

在分析过程中，编译器用了一个 **alive 变量**来代表代码是否会执行到当前位置。打开 **Flow$LiveAnalyzer 类**，你会看到 **visitMethodDef** 中的部分代码，如下所示。如果方法体里有正确的 return 语句，那么扫描完方法体以后，alive 的取值是“DEAD”，也就是这之后不会再有可执行的代码了；否则就是“ALIVE”，这意味着 AST 中并不是所有的分支，都会以 return 结束。

复制代码

```
public void visitMethodDef(JCMethodDecl tree) {
    ...
    alive = Liveness.ALIVE;  //设置为ALIVE
    scanStat(tree.body);     //扫描所有的语句

    //如果仍然是ALIVE，但返回值不是void，那么说明缺少Return语句
    if (alive == Liveness.ALIVE && !tree.sym.type.getReturnType().hasTag(VOID))
        log.error(TreeInfo.diagEndPos(tree.body), Errors.MissingRetStmt);
   ...
}
```

你可以看到下面的代码示例中，当递归下降地扫描到 if 语句的时候，只有同时存在 **then** 的部分和 **else** 的部分，并且两个分支的活跃性检查的结果都是“DEAD”，也就是两个分支都以 return 语句结束的时候，if 节点执行后 alive 就会变成“DEAD”，也就是后边的语句不会再被执行。除此之外，都是“ALIVE”，也就是 if 后边的语句有可能被执行。

复制代码

```
public void visitIf(JCIf tree) {
    scan(tree.cond);          //扫描if语句的条件部分
    //扫描then部分。如果这里面有return语句，alive会变成DEAD
    scanStat(tree.thenpart);
    if (tree.elsepart != null) {
        Liveness aliveAfterThen = alive;
        alive = Liveness.ALIVE;
        scanStat(tree.elsepart);
        //只有then和else部分都有return语句，alive才会变成DEAD
        alive = alive.or(aliveAfterThen);
    } else {  //如果没有else部分，那么把alive重新置为ALIVE
        alive = Liveness.ALIVE;
    }
}
```

看代码还是比较抽象。我把数据流分析的逻辑用控制流图的方式表示出来，你看着会更直观。

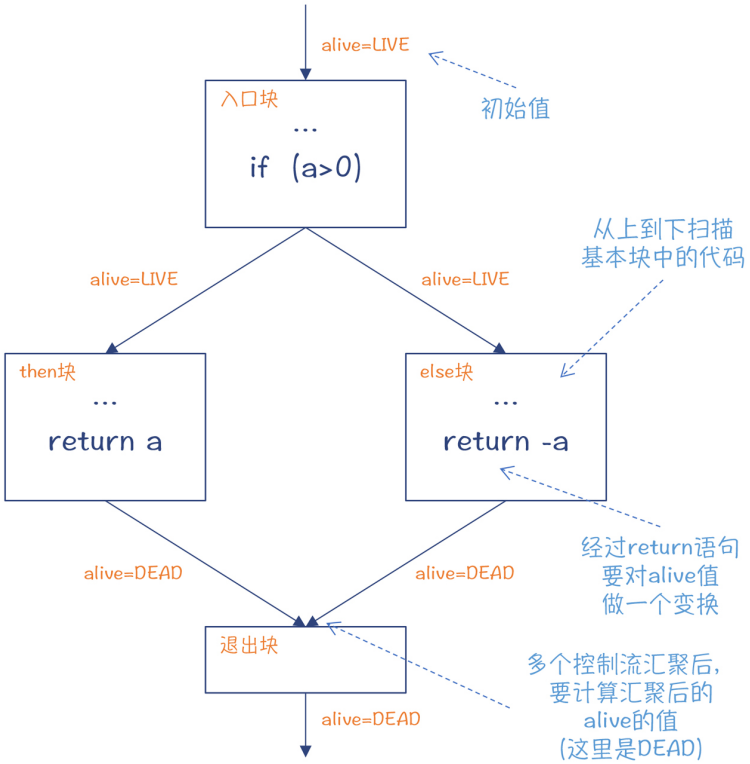


图 7：活跃性分析

我们通过活跃性分析，可以学习到数据流分析框架的 5 个要素：

1. **V**：代表被分析的值，这里是 alive，代表了控制流是否会到达这里。
2. **I**：是 V 的初始值，这里的初始值是 LIVE；
3. **D**：指分析方向。这个例子里，是从上到下扫描基本块中的代码；而有些分析是从下往上的。
4. **F**：指转换函数，也就是遇到每个语句的时候，V 如何变化。这里是在遇到 return 语句的时候，把 alive 变为 DEAD。
5. **Λ**：meet 运算，也就是当控制流相交的时候，从多个值中计算出一个值。你看看下图，在没有 else 块的时候，两条控制流中 alive 的值是不同的，最后的取值是 LIVE。

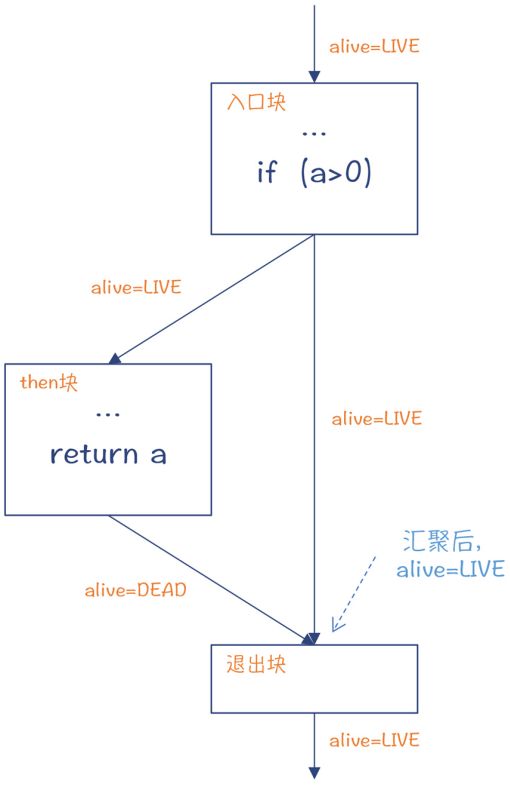


图 8：当没有 else 块的时候，两条控制流中的 alive 值不同

在做 meet 运算的时候，会用到一个叫做半格的数学工具。你可以参考本讲末尾的链接。

好了，我借助活跃性分析给你简要地讲解了数据流分析框架，我们接着往下看。

再进一步，活跃性分析还可以检测不可到达的语句。

如果我们在 return 语句后面再加一些代码，那么这个时候，alive 已经变成“DEAD”，编译器就会报“语句不可达”的错误。

Java 编译器还能检测更复杂的语句不可达的情况。比如在下面的例子中，a 和 b 是两个 final 类型的本地变量，final 修饰词意味着这两个变量的值已经不会再改变。

复制代码

```
public class Unreachable{
  public void foo(){
        final int a=1;
    final int b=2;
    while(a>b){ //a>b的值可以在编译期计算出来
      System.out.println("Inside while block");
    }
```

```
8       System.out.println("Outside while block");
9     }
10  `
```

这种情况下，在扫描 **while 语句**的时候，条件表达式“a>b”会被计算出来，是 false，这意味着 while 块内部的代码不会被执行。注意，在第 7 讲的优化算法中，这种优化叫做**稀疏有条件的常数折叠**。因为这里是用于编译器前端，所以只是报了编译错误。如果是在中后端做这种优化，就会直接把不可达的代码删除。

复制代码

```
1  //Flow$AliveAnalyzer
2  public void visitWhileLoop(JCWhileLoop tree) {
3      ListBuffer<PendingExit> prevPendingExits = pendingExits;
4      pendingExits = new ListBuffer<>();
5      scan(tree.cond);          //扫描条件
6      alive = Liveness.from(!tree.cond.type.isFalse());  //如果条件值为false,那么al
7      scanStat(tree.body);      //扫描while循环体
8      alive = alive.or(resolveContinues(tree));
9      alive = resolveBreaks(tree, prevPendingExits).or(
10         !tree.cond.type.isTrue());
11 }
12
13 void scanStat(JCTree tree) { //扫描语句
14     //如果在扫描语句的时候，alive是DEAD，那么该语句就不可到达了
15     if (alive == Liveness.DEAD && tree != null) {
16         log.error(tree.pos(), Errors.UnreachableStmt);
17         if (!tree.hasTag(SKIP)) alive = Liveness.RECOVERY;
18     }
19     scan(tree);
20 }
```

还有一种代码不可达的情况，就是无限循环后面的代码。你可以思考一下，在上面的例子中，**如果把 while 条件的“a>b”改成“a<b”，会发生什么情况呢?**

编译器会扫描 while 里面有没有合适的 break 语句（通过 **resolveBreaks() 方法**）。如果找不到，就意味着这个循环永远不会结束，那么循环体后面的语句就永远不会到达，从而导致编译器报错。

除了活跃性分析，Flow 阶段还做了其他三项分析：**异常分析、赋值分析和本地变量捕获分析。**

为了方便你的学习，我把 Java 编译器用到的几个数据流分析方法整理了一下，放在下面的表格中：

分析方法	英文	代码位置	说明
活跃性分析	Liveness Analysis	LiveAnalyzer	检查每个语句是否都是可到达的。
异常分析	Exception Analysis	FlowAnalyzer	检查每个异常要么被捕捉，要么被继续抛出。 这个分析器用到了活跃性分析的一个结果，**finallyCanCompleteNormally字段。**
赋值分析	Assignment Analysis	AssignAnalyzer	有两个赋值分析：**确定的赋值分析**（Definite Asignment Analysis，DA），检查每个变量在使用之前是否已经被赋值，Java要求在读取一个变量的值之前，它必须至少被赋值过一次；**确定的未赋值分析**（Definite Unassignment Analysis，DU），检查final类型的变量不会被赋值1次以上。 **AssignAnalyzer依赖活跃性分析的结果。**它还能分析出哪些变量实质上是final的（effectively-final），也就是只被赋值了一次。
本地变量捕获分析	Local Variable Capture Analysis	CaptureAnalyzer	确保被内部类和Lambda所访问的本地变量，要么是final的，要么实质上是final的。**它要在赋值分析之后进行。**

这几种分析方法，我建议你可以做几个例子，跟踪代码并研究一下，会加深你对数据流分析的直观理解。

异常分析、赋值分析和本地变量捕获的思路，与活跃性分析类似，它们都是按照数据流分析框架来处理的。也就是说，对于每个分析方法，你都要注意识别出它的五大要素：值、初始值、转换规则、扫描方向，以及 meet 运算规则。

课程小结

今天这一讲，我们研究了 Java 编译过程中的属性分析和数据流分析两个阶段。

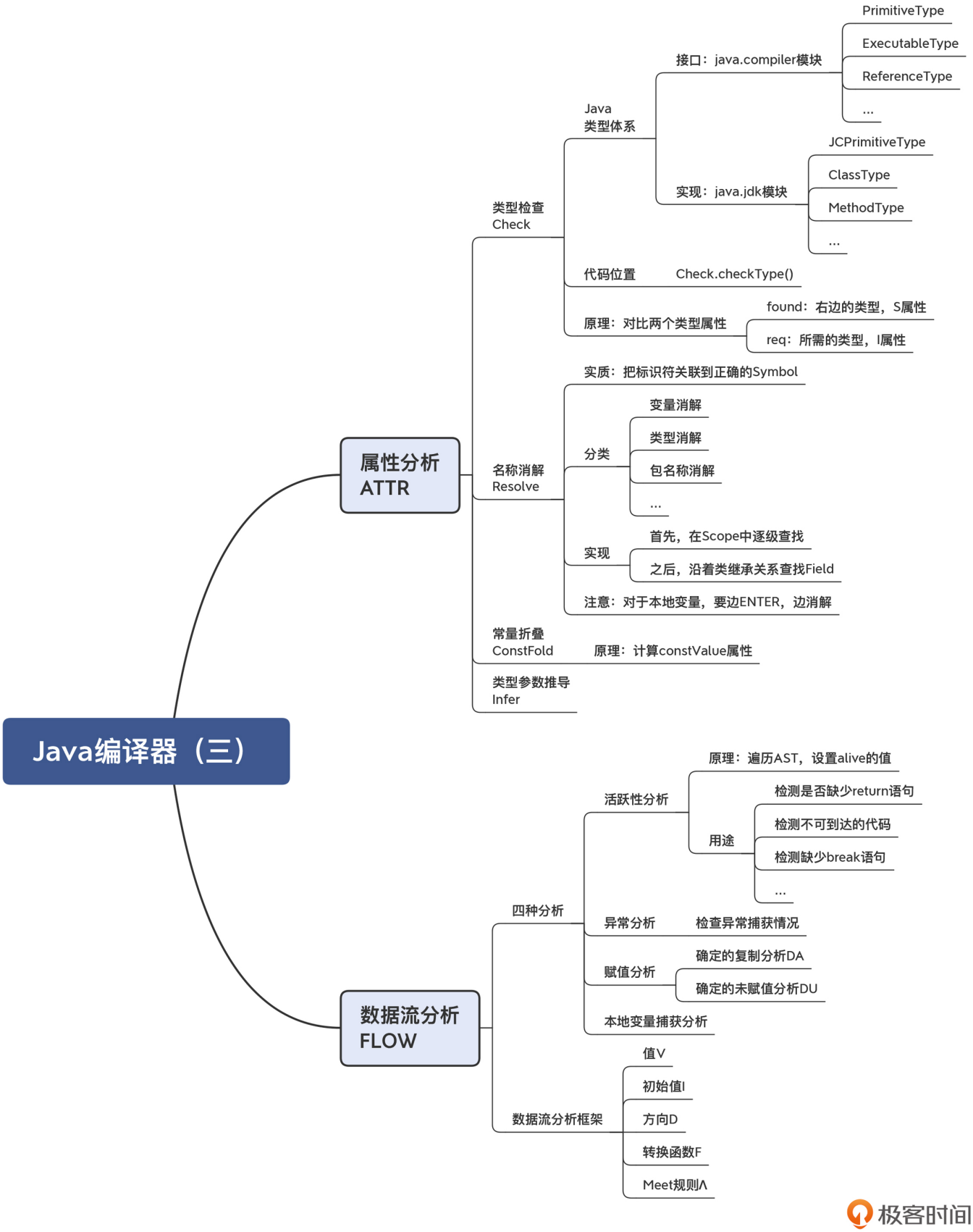
在属性分析阶段，你能够看到 Java 是如何做类型检查、引用消解、常量折叠和推导类型参数的，它们实际上是对类型（type）、符号（sym）、常量值（constValue）和类型参数这 4 类属性的处理工作。

我们也见识到了在编译器前端的**数据流分析阶段**，是如何使用数据流分析方法的。通过数据流分析，编译器能够做一些更加深入的语义检查，比如检查控制流是否都经过了 return 语

句，以及是否有不可到达的代码、每个异常是否都被处理，变量在使用前是否肯定被赋值，等等。

总体来说，在 ATTR 和 FLOW 这两个阶段，编译器完成了主要的语义检查工作。如果你在设计一门语言的时候，遇到了如何做语义检查的问题，那你就可以参考一下这一讲的内容。

在最后，是本节课程知识点的思维导图，供你参考：



一课一思

数据流分析框架很重要，你可以借助实例对它熟悉起来。那么，你能针对赋值分析，把它的 5 个元素列出来吗？欢迎在留言区分享你的思考，我会在下一讲的留言区，通过置顶的方式公布标准答案。

如果你觉得有收获，欢迎你把今天的内容分享给你的朋友。

参考资料

1. 关于数据流分析的理论性内容，可以参考龙书（Compilers Principles, Techniques and Tools）第二版的 9.2 和 9.3 节。你也可以参考《编译原理之美》 的第27、28讲，那里进行了比较直观的讲述。
2. 关于半格这个数学工具，你可以参考龙书第二版的 9.3.1 部分，也同样可以参考《编译原理之美》的第 28 讲。

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 10 | Java编译器（二）：语法分析之后，还要做些什么？

下一篇 12 | Java编译器（四）：去除语法糖和生成字节码

精选留言 (1)

写留言

宫文学Richard 置顶

2020-06-30

上一讲思考题的参考解答：
针对示例程序，Java会形成5个词法作用域：
1.类作用域
2.方法作用域，这里主要放参数
3.方法体作用域...
展开

1